Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 321 (2004) 95–101
Theoretical Computer Science
www.elsevier.com/locate/tcs

An efficient algorithm for sequence comparison with block reversals

S. Muthukrishnan[a,b,*], S. Cenk Sahinalp[c,d,e]

[a] *Department of CS, Rutgers University, USA*
[b] *AT&T Labs - Research, Florham, Park, NJ, USA*
[c] *Department of EECS, Case Western Reserve University, Cleveland, OH, USA*
[d] *Department of Genetics, Case Western Reserve University, Cleveland, OH, USA*
[e] *Center for Computational Genomics, Case Western Reserve University, Cleveland, OH, USA*

Received 10 November 2002; received in revised form 2 May 2003; accepted 30 May 2003

Abstract

Given two sequences X and Y that are strings over some alphabet set, we consider the distance $d(X,Y)$ between them defined to be minimum number of character replacements and block (substring) reversals needed to transform X to Y (or vice versa). The operations are required to be disjoint. This is the "simplest" sequence comparison problem we know of that allows natural block edit operations. Block reversals arise naturally in genomic sequence comparison; they are also of interest in matching music data. We present an algorithm for exactly computing the distance $d(X,Y)$; it takes time $O(|X|\log^2|X|)$, and hence, is near-linear. Trivial approach takes quadratic time.

Keywords: Sequence comparison; Block edit distance; String periodicity

1. Introduction

Computing the distance between two sequences under edit operations is a central problem in combinatorial pattern matching. The distance $d(X,Y)$ between sequences X and Y can be defined as the minimum number of permitted edit operations needed to transform one to another (all edit operations of interest are reversible so that $d(X,Y)=d(Y,X)$ for any two sequences X and Y). The goal is to compute $d(X,Y)$

* Corresponding author. Department of EECS, Case Western Reserve University, Cleveland, OH, USA. Tel.: +1-216-368-6197; fax: +1-216-368-2801.
E-mail addresses: muthu@cs.rutgers.edu (S. Muthukrishnan), cenk@cwru.edu (S. Cenk Sahinalp).

0304-3975/$ - see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.tcs.2003.05.005

for given X and Y as efficiently as possible. The nature of the distance computation problem depends on the edit operation that is permitted, which corresponds to the notion of similarity between sequences that we wish to capture for the application in question.

Natural edit operations include character edits and block edits.

Character edits include

(1) inserting a single character to any specified location,
(2) deleting any given character, and
(3) replacing a single character by another.

Among these edit operations, character replacements are particularly common in computational genomics, in the form of mutations in biomolecular sequences. Sequence distances involving character replacements are used to estimate the evolutionary distances between DNA, RNA or protein sequences [12], construct phylogenetic trees, search for common motifs in the genome with functional homology, etc. [5]. Insert and delete operations are also quite commonly observed especially as sequencing errors in computational genomics [3], transcription errors in text, transmission errors in communication channels, etc. These are of classical interest, having been studied at least as early as 1960s [7].

Block (substring) edits include

(1) reversing a block (commonly observed in genomic sequences, usually as a consequence of large scale inter or intra chromosomal genomic duplications; also common in multimedia data such as music scales [11]),
(2) copying a block from one location to another (another genomic phenomena commonly observed in the form of genomic duplications and rearrangements [5]),
(3) deleting a copy of a block that exists elsewhere (motivated by general macro compression schemes [13], especially in the form of Ziv–Lempel-type data compression [16]),
(4) moving a block from one location to another (due to moving a paragraph of a text to another location [14], or moving objects around in pen computing [8]). [1]

Although these edit operations are motivated by many applications, we do not consider the specifics of any application. Rather, we focus on a central complexity question, namely, how to compute the various character and block edit distances efficiently.

The main result of this paper is on the basic problem of computing the distance between two sequences with both character and block edit operations. If the distance $d(X, Y)$ between two sequences X and Y is defined so as to allow any block copy, delete or move operations the problem of computing $d(X, Y)$ becomes NP-hard [8]. Thus, among the block edit operations described above, we allow only block reversals. Since, it is not possible to transform a given sequence X (e.g., all 0's) to every other sequence Y (e.g., all 1's) by block reversals alone, $d(X, Y)$ is not well defined when only block reversals are allowed. Therefore, the "simplest" well-defined distance with

[1] There are other block edit operations of interest, such as *block linear transformations* which involves changing each position of $Q[i]$ in a block Q, to $\phi Q[i] + \kappa$, where ϕ and κ are respective multiplicative and additive scaling constants. Such edit operations are quite common in financial data analysis where one allows scaling effects in tracking similar trends [1].

block reversals additionally allows character replacements. Henceforth, we will focus on this distance $d(X,Y)$ between two sequences X and Y formally defined to be the minimum number of character replacements and block reversals needed to convert X to Y (equivalently, vice versa). We require that these operations be disjoint, that is, no location in X or Y is the target of more than one operation. Observe that $d(X,Y)$ is only defined when $|X| = |Y|$.

Using dynamic programming, we can compute $d(X,Y)$ exactly in time $\Theta(|X|^2)$; the solution is straightforward. Of course, it is trivial to compute the distance in linear time if only character replacements are considered and block reversals are *not* allowed. The first nontrivial algorithm for solving this problem was presented in [9] which took time $O(|X| \log^3 |X|)$. An improvement to $O(|X| \log^2 |X|)$ time was presented in [], which we present here. Subquadratic algorithms are in general of interest in computing various edit distances since the basic problem of computing character edit distance is not known to have a near-linear time solution at present.

2. Preliminaries

In the rest of the paper we denote sequences by $P, Q, R, S \ldots$, integers by $i, j, k \ldots$ (represented in binary) and constants by $\alpha, \beta, \gamma \ldots$. All sequences have characters drawn from an (integer size) alphabet σ. Given a sequence S, let $|S|$ denote its length and $S[i]$ denote its ith character. A *block* of S will be any subsequence $S[i:j]$ extending from $S[i]$ to $S[j]$. We denote by $S||Q$ the concatenation of two sequences S and Q, and by Q^l, a sequence obtained by concatenating l copies of Q. The *reverse* of S is the sequence $S[|S|], \ldots, S[2], S[1]$, and is denoted by $S^{\leftarrow}$. A sequence S is called *periodic* with $l > 1$ repetitions if there exists a block Q such that $S = Q^l||Q^-$, where Q^- is a prefix of Q; Q is called a *period* of S. The shortest such Q is called *the* period of S; the period of S is the period of all periods of S.

3. The main result

Let X and Y be two size ℓ sequences. Recall that $d(X,Y)$ is the minimum number of disjoint character replacements and block reversals needed to transform X into Y. One can use standard dynamic programming together with block labeling to compute $d(X,Y)$ in $O(\ell^2)$ time. Below we describe an algorithm for computing $d(X,Y)$ in time $O(\ell \log^2 \ell)$. The procedure relies on combinatorial properties of reversals in sequences.

We say that block $X[i:j]$ is *reversible* (with respect to Y) if $X[i:j] = Y[i:j]^{\leftarrow}$.

3.1. The algorithm

We compute $d(X,Y)$ in ℓ iterations as follows. In iteration i, we compute $d(i) = d(X[1:i], Y[1:i])$.

Let $X[i_1 : i]$ be the longest suffix of $X[1:i]$ that is reversible and let p_1 be the length of the period of $X[i_1 : i]$. Let $\#_1$ to be the number of occurrences of p_1 in

$X[i_1 : i]$. We define $i_2 = i_1 + \#_1 \cdot p_1$. In other words $X[i_2 : i]$ is the suffix which remains when the maximum possible number of periods are removed from the prefix of $X[i_1 : i]$.

We can now inductively define the following for $h \geqslant 1$:
$\#_h = \lceil \frac{i - i_h + 1}{p_h} \rceil - 1$,
$i_{h+1} = i_h + \#_h \cdot p_h + 1$,
p_{h+1}: the length of the period of $X[i_{h+1} : i]$.
The reader can observe that $X[i_{h+1} : i]$ is the remaining suffix when maximum possible number of periods of $X[i_h : i]$ (i.e. $\#_h$) are removed from its prefix.

Let H be the minimum h such that $\#_h = 0$. Then H is upper bounded by $\log i$:

Lemma 1. $H \leqslant \log i$.

Proof. Notice that $\#_h$ is either 0 or is at least 2: by definition, a period must occur at least twice. Therefore $i - i_h > 2(i - i_{h+1})$ and thus $H \leqslant \log i - i_1 \leqslant \log i$. □

The algorithm computes $d(i)$ as follows. For $1 \leqslant h \leqslant H$, we set

$$d_h(i) = \begin{cases} d(i_h - 1) + 1 & \text{if } \#_h \leqslant 1, \\ d_h(i - p_h) & \text{if } \#_h > 1. \end{cases}$$

We also set $d_0(i) = d(i-1) + d(X[i], Y[i])$. Then we compute $d(i) = \min\{d_h(i)\}$ which completes the description of the algorithm.

We prove the correctness of the algorithm through the following lemma and its corollaries.

Lemma 2. *If $X[j : i]$ is reversible and p is the length of the period of $X[j : i]$, then for any $k \leqslant i - j - p + 1$, the substring $X[j + k : i]$ is reversible if and only if $k = p \cdot h$ for $0 \leqslant h \leqslant \lceil (k + 1)/p \rceil - 1$.*

Proof. (1) If $X[j : i]$ is reversible, then $X[j : i] = Y[j : i]^{\leftarrow}$, and hence $X[j+k] = Y[i-k]$ for all $k \leqslant j - i + 1$. Because p is the length of the period of $X[j : i]$, $X[j+k] = X[j+p \cdot h+k] = Y[i-k]$ for all $k < i - j - (p \cdot h) + 1$, which implies $X[j+p \cdot h : i] = Y[j+p \cdot h : i]^{\leftarrow}$; therefore $X[j + p \cdot h : i]$ is reversible.

(2) If both $X[j : i]$ and $X[j+k : i]$ are reversible, then $X[j+h] = Y[i-h] = X[j+k+h]$ for all $h \leqslant i - j - k$. Therefore k must be the length of a period of $X[j : i]$, which implies that k must be a multiple of p. □

The statements below follow.

Corollary 3. (i) *The only reversible suffixes of $X[1 : i]$ are of the form $X[i_h + p_h \cdot j : i]$ for all $0 \leqslant j \leqslant \#_h$ and $1 \leqslant h \leqslant H$.*

(ii) *If $X[j : i]$ is reversible and p is the length of the period of $X[j : i]$, then for any $k \geqslant j + p$, the substring $X[j : k]$ is reversible if and only if $k = j + p \cdot h$ for $0 \leqslant h \leqslant \lceil (k - j + 1)/p \rceil - 1$.*

(iii) *Let* $X[j:i]$ *be a reversible string whose period is of length* p *and let* $X[h: i-k\cdot p]$ *be a substring of* $X[j:i]$ *such that* $h+p<i-k\cdot p$. *If* $X[h:i-k\cdot p]$ *is reversible then so is* $X[h:i]$.

Thus the proof of the lemma below follows, immediately giving the correctness of the entire algorithm.

Lemma 4. *Consider any* i *and let* $p_0 = i - i_1$, *where* i_1 *is as defined in the algorithm. For* $1 \leqslant h \leqslant H$, $d_h(i)$ *is equal to the distance between* $X[1:i]$ *and* $Y[1:i]$, *provided a suffix of length* k *for which* $p_{h-1} \geqslant k > p_h$ *is reversed* (*while transforming* $X[1:i]$ *to* $Y[1:i]$).

It remains for us to show how to implement the algorithm, and determine its running time.

We first compute i_1 for all i, $1 \leqslant i \leqslant \ell$ in $O(\ell)$ time as follows.

(1) First we set $X' = X[1], \$, X[2], \$, \ldots, \$, X[\ell]$ and $Y' = Y[1], \$, Y[2], \$ \ldots, \$, Y[\ell]$ where \$ is a special character which is not in the original alphabet and used for treating reversals whose center is between two characters and those whose center is a character uniformly. Notice that for $X[i:j]$ is reversible with respect to Y if and only if $X'[2i-1:2j-1]$ is reversible with respect to Y'.

(2) For all $1 \leqslant j \leqslant 2\ell - 1$ we compute the largest k for which $X'[j-k:j+k]$ is reversible with respect to Y'; let this be k_j. This can be done by finding the longest common prefix between $X'[j:2\ell-1]$ and $Y'[1:j]^{\leftarrow}$, the longest common suffix of $X'[1:j]$ and $Y'[j:2\ell-1]^{\leftarrow}$, and picking the shortest of the two. For finding either of the two longest common items, it suffices to build the joint suffix trees of X and $Y'^{\leftarrow}$; this takes $O(\ell)$ time to construct [15,2]. This joint suffix tree can then be preprocessed in $O(\ell)$ time in order to answer the lowest common ancestor queries in time $O(1)$ each [4].

(3) For all $i = 1, \ldots, 2\ell - 1$, notice that i_1 is the equal to the smallest j for which $j + k_j \geqslant i$. We compute i_1 for all i in $O(\ell)$ time due to the observation that for $i < i'$, $i_1 \leqslant i'_1$. Thus for each i, we only consider $j = (i-1)_1, (i-1)_1 + 1, \ldots$, until we find $j = i_1$.

3.1.1. Computing the periods of substrings

Next, we focus on computing the period of any substring $S = X[j:i]$. We show that one can preprocess X in $O(\ell \log \ell)$ time to be able to compute the period of any given substring $S = X[j:i]$ in $O(\log |S|)$ time.

(1) *Preprocessing*: We simply assign labels to all substrings of X of size 2^k, $0 \leqslant k \leqslant \lfloor \log \ell \rfloor$, such that (i) each distinct substring gets a distinct label, and (ii) each substring has a pointer to the nearest substring to its left which has the same label. This can be done in $O(\ell \log \ell)$ time using the labeling technique described in [6]. We use these labels to answer two types of queries:

(a) Checking whether any two substrings R and Q (of any given length i) are identical: Q and R are identical if and only if their length $2^{\lfloor \log i \rfloor}$ prefixes are identical and their length $2^{\lfloor \log i \rfloor}$ suffixes are identical. Thus to check whether Q and R are identical, all we need to do is to check whether these suffixes and prefixes are identical by simply comparing their respective labels in $O(1)$ time.

(b) Checking whether S has a period of length j for any specified length $j \leqslant |S|/2$: S has a period of length j if and only if $S[1 : |S| - j]$ is identical to $S[j + 1 : |S|]$. This can be done by the simple method described above in O(1) time.

Querying: Using the labels obtained in the preprocessing step we compute the period of any given substring $S = X[j : i]$ in $\log |S|$ iterations. In the kth iteration ($k = 0, 1, \ldots, \lceil \log |S| \rceil - 1$) we decide either (i) S does not have a period of length in the range $2^{k-1} + 1, \ldots, 2^k$ and thus S does not have a period shorter than 2^k), or (ii) S has a period of length in the range $2^k, \ldots, 2^{k-1} + 1$, and explicitly compute the length of the period, or (iii) S is non-periodic. This is done in O(1) time per iteration as follows. We first check if there exists 0, 1 or more *candidate* period lengths l in the range $2^{k-1} + 1, \ldots, 2^k$: a candidate period length l is one which satisfies the condition that $S[|S| - l - 2^{\lceil \log |S| \rceil - 1} + 1 : |S| - l]$ is identical to $S[|S| - 2^{\lceil \log |S| \rceil - 1} + 1 : |S|]$. This can be done in O(1) time by (i) following the pointer of $X[i - 2^{\lceil \log |S| \rceil - 1} + 1 : i]$ to the nearest substring to its left with an identical label, (ii) checking if the rightmost position of this substring is in the range $i - 2^k, \ldots, i - 2^{k-1}$ (if yes, there is at least one candidate substring), and (iii) following the pointer of this substring to a new substring and checking whether this new one has its rightmost position in the same range (if yes, there are at least two candidates).

Notice that if l is a period length of S then $S[1 : |S| - l]$ and $S[l + 1 : |S|]$ should be identical, and so as their equal length prefixes. Thus if l is the length of a period of S then it should be identified as a candidate period length according to our definition.

(a) If no such candidate period length exists, then S cannot have a period shorter than $2^k + 1$ and thus we move to the next iteration.

(b) If only one candidate period length l exists, then l is the only possible period length for S in the range $2^{k-1}, \ldots, 2^k$. We simply check in O(1) time whether l is a period length of S via the simple test described in the preprocessing stage.

(c) If two (or more) such candidate period lengths exist, then we conclude that S is non-periodic due to the following observation. If the length of the period of S is p_S and if for any of its substrings, R, the length of its period is p_R, then either $p_S = p_R$ or $p_S \geqslant |R|$.

Now consider two candidate period lengths $l_1 \leqslant l_2$. We know that $R_1 = S[|S| - l_1 - 2^{\lceil \log |S| \rceil - 1} + 1 : |S| - l_1]$ is identical to $S[|S| - 2^{\lceil \log |S| \rceil - 1} + 1 : |S|]$ which in turn is identical to $R_2 = S[|S| - l_2 - 2^{\lceil \log |S| \rceil - 1} + 1 : |S| - l_2]$; thus the length of the period of $R = S[|S| - l_2 - 2^{\lceil \log |S| \rceil - 1} + 1 : |S| - l_1]$ (the concatenation of R_1 and R_2) is $p_R = l_2 - l_1 \leqslant 2^{k-1}$. We inductively know that $p_S > 2^{k-1} \geqslant p_R$, thus it must be the case that $p_S \geqslant |R| > 2^{\lceil \log |S| \rceil - 1}$ which means that S is non-periodic.

This combined with the iteration for computing $d(i)$'s gives our main result.

Theorem 5. *Given two sequences X and Y of length ℓ, there exists an* $O(\ell \log^2 \ell)$ *time algorithm to compute* $d(X, Y)$.

4. Discussion

We have given a near linear time algorithm to compute $d(X, Y)$ which is the minimum number of character replacements and block reversals needed to convert X and Y

(and vice versa). The function $d()$ is the "simplest" distance involving block edit operations, and this paper gives the fastest algorithm for computing any block edit distance. It will be of interest to design efficient algorithms for simple sequence comparison problems with other block operations [8].

References

[1] R. Agarwal, K. Lin, H. Sawhney, K. Shim, Fast similarity search in the presence of noise, scaling and translation in time-series databases, Proc. 21st VLDB Conf, 1995.
[2] M. Farach-Colton, Optimal suffix tree construction with large alphabets, IEEE FOCS, 1997.
[3] M. Gribskov, J. Devereux, Sequence Analysis Primer, Stockton Press, NY, 1991.
[4] D. Harel, R. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.
[5] M. Jackson, T. Strachan, G. Dover, Human Genome Evolution, Bios Scientific Publishers, Oxfordshire, 1996.
[6] R. Karp, R. Miller, A. Rosenberg, Rapid identification of repeated patterns in strings, trees, and arrays, Proc. ACM Symp. on Theory of Computing, 1972.
[7] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, Cybernet. Control Theory 10 (8) (1966) 707–710.
[8] D. Lopresti, A. Tomkins, Block edit models for approximate string matching, Theoret. Comput. Sci. 181 (1) (1997) 159–179.
[9] S. Muthukrishnan, S.C. Sahinalp, Approximate nearest neighbors and sequence comparison with block operations, Proc. ACM Symp. on Theory of Computing, 2000.
[10] S. Muthukrishnan, S.C. Sahinalp, An improved algorithm for sequence comparison with block reversals, Proc. LATIN, 2002.
[11] D. Sankoff, J. Kruskal, Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, Reading, MA, 1983.
[12] P. Sellers, The theory and computation of evolutionary distances: pattern recognition, J. Algorithms 1 (1980) 359–373.
[13] J.A. Storer, Data Compression, Methods and Theory, Computer Science Press, Rockville, MD, 1988.
[14] W.F. Tichy, The string-to-string correction problem with block moves, ACM Trans. Comput. Systems 2 (4) (1984) 309–321.
[15] P. Weiner, Linear pattern matching algorithms, Proc. IEEE Foundations of Computer Science (FOCS), 1973, pp. 1–11.
[16] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory (1977) 337–343.